An Autonomic Hierarchical Reliable Broadcast Protocol for Asynchronous Distributed Systems with Failure Detector

Denis Jeanneau*, Luiz A. Rodrigues†, Luciana Arantes* and Elias P. Duarte Jr. ‡
* *Sorbonne Universités, UPMC Univ. Paris 06,CNRS, Inria, LIP6, Paris, France*
E-mail: firstname.lastname@lip6.fr
†*Western Paraná State University, Department of Computer Science, Cascavel, Brazil*
E-mail: luiz.rodrigues@unioeste.br
‡*Federal University of Paraná, Department of Informatics, Curitiba, Brazil*
E-mail: elias@inf.ufpr.br

***Abstract*—Reliable broadcast protocol is a fundamental building block in fault-tolerant distributed systems. It consists of a basic primitive that provides agreement among processes of the system on the delivery of each broadcast message, i.e., either none or all correct processes deliver the message, despite failures of processes. In this work, we propose a reliable broadcast solution on top of the VCube, assuming that the system is asynchronous. The VCube is an autonomic monitoring layer that organizes processes on a hypercube overlay which provides several logarithmic properties even in the presence of processes failures. We consider that processes fail by crashing, do not recover, and faults are eventually detected by all correct processes. The protocol tolerates false suspicions by sending additional messages to suspected processes but logarithmic properties of the algorithm are still kept.**

***Index Terms*—Implementation of Distributed Systems, Asynchronous system, Autonomic Computing, Fault-Tolerant Broadcasts, Spanning trees**

1. Introduction

Numerous distributed applications with information dissemination requirements rely on a broadcast communication primitive to send messages to all processes that compose the application [1]. Formally, reliable broadcast is defined in terms of two primitives: *broadcast(m)*, which is defined by the broadcast algorithm and called by the application to disseminate m to all processes, and *deliver(m)*, which is defined by the application and called by the broadcast algorithm when message m has been received. The broadcast algorithm that offers these primitives must ensure that, if a correct[1] process broadcasts a message, then it eventually delivers the message (*validity* property). Furthermore, every correct process delivers a message at most once and only if that message was previously broadcast by some process (*integrity* property).

From an implementation point of view, the broadcast primitive sends point-to-point messages to each process of the system. However, if the sender fails during the execution of the broadcast primitive, some processes might not receive the broadcast message. In order to circumvent this problem, *reliable broadcast* ensures, besides the *validity* and *integrity* properties, that even if the sender fails, every correct process delivers the same set of messages (*agreement* property) [2].

There exists a considerable amount of literature on reliable broadcast algorithms, such as the one where all correct receivers retransmit all received messages guaranteeing then the delivery of all broadcast messages by the other correct processes of the system [3]. We are particularly interested in solutions that use failure detectors [4] which notify the broadcast algorithm about processes failures. Upon receiving such an information, the algorithm reacts in accordance to tolerate the failure. Another important feature of reliable broadcast algorithms concerns performance, which is related to how broadcast messages are diffused to processes. Aiming at scalability and message complexity efficiency, many reliable broadcasts organize processes on logical spanning trees. Messages are then diffused over the constructed tree, therefore providing logarithmic performance [5], [6], [7], [8], [9] (see Section 2).

This work presents an autonomic reliable broadcast algorithm where messages are transmitted over spanning trees dynamically built on top of a logical hierarchical hypercube-like topology. Autonomic systems constantly monitor themselves and automatically adapt to changes [10]. The logical topology is maintained by the underlying VCube monitoring system which also detects failures [11] [12]. VCube is a distributed diagnosis layer responsible for organizing processes of the system in a virtual hypercube-like cluster-based topology which is dynamically re-organized in case of process failure. When invoked, the VCube gives information about the liveness of the processes that compose the system.

We assume a fully-connected **asynchronous** system in which processes can fail by crashing, and crashes are permanent. Links are reliable. A process that invokes the reliable broadcast primitive starts the construction of a spanning tree. This tree is built with information obtained from the VCube, and is dynamically reconstructed upon detection of a node crash (process failure).

[1]A correct process is a process that does not fail during execution

In a previous work [13], we proposed an autonomic reliable broadcast algorithm on top of the Hi-ADSD, a previous version of the VCube. The algorithm guarantees several logarithmic properties, even when nodes fail, and allows transparent and efficient spanning tree reconstructions. However, for this solution, we considered a synchronous model for the system, i.e., there exist known bounds on message transmission delays and processors' speed and, consequently, the VCube needs to provide perfect process failure detections. On the one hand, the advantage of such synchronous assumption is that there was no false failure suspicions and, thus, if the VCube notifies the broadcast algorithm that a given process is faulty, the algorithm is sure that it can stop sending message to this faulty process and then removes it forever from the spanning tree constructions. On the other hand, the synchronous assumption considerably restrains the distributed systems and applications that can use the broadcast protocol since many of the current network environments are considered asynchronous (there exist no bounds on message transmission delay or on processors' speed).

Hence, considering the above constraints, we propose in this article a new autonomic reliable broadcast algorithm, using the VCube in an asynchronous model. We assume that the failure detection service provided by the VCube is unreliable since it can make mistakes by erroneously suspecting a correct process (false suspicion) or by not suspecting a node that has actually crashed. However, upon detection of its mistake, the VCube corrects it. Furthermore, it also ensures that eventually all failures are detected (*strong completeness* property). Note that such false suspicions render a broadcast algorithm much more complex than the previous one since it can induce violation of the properties. For instance, the algorithm must ensure that a falsely suspected process must receive and deliver, only once, all broadcast messages, otherwise the *agreement* and *integrity* properties would be violated. In our solution, false suspicions are tolerated by sending special messages to those processes suspected of having failed. We must also emphasize that our aim is to provide a reliable broadcast algorithm which is efficient, i.e., that keeps, as much as possible, the logarithmic properties of the spanning tree diffusion over the hypercube-like topology. Our algorithm tolerates up to n-1 node crashes.

The rest of this paper is organized as follows. Section 2 discusses some related work. In Section 3 we describe the system model while Section 4 briefly describes the VCube diagnosis algorithm and the hypercube-like topology. In Section 5, we present the autonomic reliable broadcast algorithm for asynchronous systems while Section 6 discuss some performance issue of the algorithm. Finally, Section 7 concludes the paper.

2. Related Work

Many reliable broadcast algorithms of the literature exploit spanning trees such as [5], [6], [7], [8], [9].

Schneider et. al. introduced in [5] a tree-based fault-tolerant broadcast algorithm whose root is the process that starts the broadcast. Each node forwards the message to all its successors in the tree. If one process p that belongs to the tree fails, another process assumes the responsibility of retransmitting the messages that p should have transmitted if it were correct. Like to our approach, processes can fail by crashing and the crash of any process is detected after a finite but unbounded time interval by a failure detection module. However, the authors do not explain how the algorithm rebuilds or reorganizes the tree after a process failure.

In [6], a reliable broadcast algorithm is provided by exploiting disjoint paths between pairs of source and destination nodes. Multiple-path algorithms are particularly useful in systems that cannot tolerate the time overhead for detecting faulty processors, but there is an overhead in the number of duplicated messages. On a star network with n edges, the algorithm constructs $n-1$ directed edge-disjoint spanning trees. Fault tolerance is achieved by retransmitting the same messages through a number of edge-disjoint spanning trees. The algorithm tolerates up to $n-2$ failure of nodes or edges and can be adjusted depending on the network reliability. Similarly, Kim et al. propose in [7] a tree-based solution to disseminate a message to a large number of receivers using multiple data paths in a context of time-constrained dissemination of information. Thus, arguing that reliable extensions using ack-based failure recovery protocols cannot support reliable dissemination with time constraints, the authors exploit the use of multiple data paths trees in order to conceive a fast and reliable multicast dissemination protocol. Basically the latter is a forest-based (multiple parents-to-multiple children) tree structure where each participant node has multiple parents as well as multiple children. A third work that exploits multi-paths spanning trees is [8] where the authors present a reliable broadcast algorithm that runs on a hypercube and uses disjoint spanning trees for sending a message through multiple paths.

Raynal et. al. proposed in [9] a reliable tree-based broadcast algorithm suited to dynamic networks in which message transfer delays are bounded by a constant of δ unit of times. Whenever a link appears, its lifetime is at least δ units of time. The broadcast is based on a spanning-tree on top of which processes forward received messages to their respective neighbors. However, as the system is dynamic, the set of current neighbors of a process p may consists of a subset of all its neighbors and, therefore, p has to additionally execute specific statements when a link re-appears, i.e., forwards the message on this link if it is not sure that the destination process already has a copy of it.

Similarly to our approach, many existing reliable broadcast algorithms exploit spanning trees constructed on hypercube-like topologies [8], [14], [15]. In [14], the authors present a fault-tolerant broadcast algorithm for hypercubes based on binomial trees. The algorithm can recursively regenerate a faulty subtree, induced by a faulty node, through one of the leaves of the tree. On the other hand, unlike our approach, there is a special message for advertising that the tree must be reconstructed and, in this case, broadcast

messages are not treated by the nodes until the tree is rebuilt. The HyperCast protocol proposed by [15] organizes the members of a multicast group in a logical tree embedded in a hypercube. Labels are assigned to nodes and the one with the highest label is considered to be the root of the tree. However, due to process failures, multiple nodes may consider themselves to be the root and/or different nodes may have different views of which node is the root.

Leitão et al. present in [16] the *HyParView*, a hybrid broadcast solution that combines a tree-based strategy with a gossip protocol. A broadcast tree is created embedded on a gossip-based overlay. Broadcast is performed by using gossip on the tree branches. Later, some of the authors proposed a second work [17] where they introduced *Thicket*, a decentralized algorithm to build and maintain multiple trees over a single unstructured P2P unstructured overlay for information diffusion. The authors argue that multiple trees approach allow that each node to be an internal node in just a few trees and a leaf node in the remaining of the trees providing, thus, load distribution as well as redundant information for fault-tolerance.

In [13], we presented a reliable broadcast solution based on dynamic spanning trees on top of the Hi-ADSD, a previous version of the VCube. Multiple trees are dynamically built, including all correct nodes, where each tree root corresponds to the node that called a broadcast primitive. Contrarily to the current work, this solution considers that the system model is synchronous and that the VCube offers a perfect failure detection.

3. System Model

We consider a distributed system that consists of a finite set P of $n > 1$ processes. Each process has a unique address. Processes $\{p_0, .., p_{n-1}\}$ communicate only by message passing. Each single process executes one task and runs on a single processor. Therefore, the terms node and process are used interchangeably in this work.

The system is asynchronous, i.e., relative processor speeds and message transmission delay are unbounded. Links are reliable, and, thus, messages exchanged between any two correct processes are never lost, corrupted or duplicated. There is no network partitioning.

Processes communicate by sending and receiving messages. The network is fully connected: each pair of processes is connected by a bidirectional point-to-point channel. Processes are organized in a virtual hypercube-like topology, called VCube. In a d-dimensional hypercube (d-cube) each process is identified by a binary address i_{d-1}, i_{d-2}, i_0. Two processes are connected if their addresses differ by only one bit. Processes can fail by crashing and, once a process crashes, it does not recover. If a process never crashes during the run, it is considered *correct* or *fault-free*; otherwise it is considered to be *faulty*. After any crash, the topology changes, but the logarithmic properties of the hypercube are kept.

We consider that the primitives to send and receive a message are atomic, but the broadcast primitives are not.

4. The VCube

Let n be the number of processes in the system P. VCube [11] is a distributed diagnosis algorithm that organizes the correct processes of the system P in a virtual hypercube-like topology. In a hypercube of d dimensions, called d-VCube, there are 2^d processes. A process i groups the other $n-1$ processes in $\log_2 n$ clusters, such that cluster number s has size 2^{s-1}. The ordered set of processes in each cluster s is denoted by $c_{i,s}$ as follows, in which $\oplus$ denotes the bitwise exclusive *or* operator (xor).

$$c_{i,s} = \{i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1}\} \qquad (1)$$

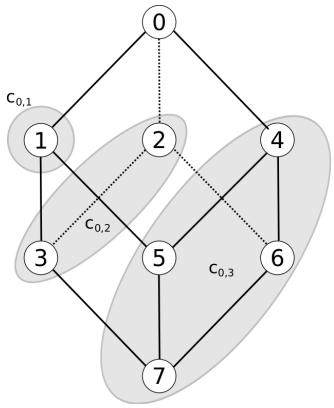
A process i tests another process in the $c_{i,s}$ to check whether it is correct or faulty. It executes a test procedure and waits for a reply. If the correct reply is received within an expected time interval, the monitored process is considered to be alive. Otherwise, it is considered to be faulty. We should point out that in an asynchronous model, which is the case in the current work, VCube provides an ureliable failure detection since it can erroneously suspect a correct process (false suspicion). If later it detects its mistake, it corrects it. On the other hand, according to the properties proposed by Chandra and Toueg [4] for unreliable failure detectors, the VCube ensures the *strong completeness* property: eventually every process that crashes is permanently suspected by every correct process. Since there are false suspicions, the VCube does not provide any *accuracy* property. A VCube providing both *completeness* and *accuracy* could not possibly be implemented in a fully asynchronous system, according to Fischer, Lynch and Paterson [18].

Timestamps are used to identify the latest state of the tested processes. Based on the replies of the tests, process i connects itself to one fault-free process of each cluster s, if it exists. If there are no failures, a complete logical hypercube is created.

Fig. 1 shows the hierarchical cluster-based logical organization of $n = 8$ processes connected by a 3-VCube topology as well as a table which contains the composition of all $c_{i,s}$ of the 3-VCube.

Let's consider process p_0 and that there are no failures. The clusters of p_0 are shown in the same figure. Each cluster $c_{0,1}$, $c_{0,2}$, and $c_{0,3}$ is tested once, i.e., p_0 only performs tests on nodes $1, 2, 4$ which will then inform p_0 about the state of the other nodes of the respective cluster.

In order to avoid that several processes test the same processes in a given cluster, process i executes a test on process $j \in c_{i,s}$ only if process i is the first faulty-free process in $c_{j,s}$. Thus, any process (faulty or fault-free) is tested at most once per round, and the latency, i.e., the number of rounds required for all fault-free processes to identify that a process has become faulty is $\log_2 n$ in average and $\log_2^2 n$ rounds in the worst case.



s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figure 1. VCube hierarchical organization.

5. Reliable Broadcast Algorithm for Asynchronous System

A reliable broadcast algorithm ensures that the same set of messages is delivered by all correct processes, even if the sender fails during the transmission. Reliable broadcast presents three properties [3]:

- *Validity*: if a correct process broadcasts a message m, then it eventually delivers m.
- *Integrity*: every correct process delivers the same message at most once (no duplication) and only if that message was previously broadcast by some process (no creation).
- *Agreement*: if a message m is delivered by some correct process p_i, then m is eventually delivered by every correct process p_j. Note that the agreement property still holds if m is not delivered by any process.

Our reliable broadcast algorithm exploits the virtual topology maintained by VCube, whenever possible. Each process creates, thus, a spanning tree rooted at itself to broadcast a message. The message is forwarded over the tree and, for every message that a node of the tree sends to one of its correct neighbor, it waits for the corresponding acknowledge from this neighbor, confirming the reception of the message. Algorithm 1 presents the pseudo-code of our proposal reliable broadcast protocol for an asynchronous system with n=2^d processes. The dimension of the VCube is, therefore, d. A process gets information, not always reliable, about the liveness of the other processes by invoking the VCube. Hence, the trees are dynamically built and autonomically maintained using the hierarchical cluster structure and the knowledge about faulty (or falsely faulty suspected) nodes. The algorithm tolerates up to n-1 failures.

Let i and j be two different processes of the system. The function $cluster_i(j) = s$ returns the identifier s of the cluster of process i that contains process j, $1 \leq s \leq d$. For instance, in the 3-cube as shown in Fig. 1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$ and $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$.

5.1. Message types and local variables

Let m be the application message to be transmitted from a sender process, denoted *source*, to all other processes in the system. We consider three types of messages:

- $\langle TREE, m\rangle$: message broadcast by the application that should be forwarded over the VCube to all processes considered to be correct by the sender;
- $\langle DELV, m\rangle$: message sent to processes suspected of being faulty in order to avoid that false suspicions induce the no delivery of the message by correct processes. The recipient of the message should deliver it but not forward it;
- $\langle ACK, m\rangle$: used as an acknowledgement to confirm that a TREE message related to m was received.

For the sake of simplicity, we use TREE, DELV, and ACK to denote these messages.

Every message m keeps two parameters: (1) the identifier of the process that broadcast m and (2) a *timestamp* generated by the process local counter which uniquely identifies the m. The first message broadcast by a process i has timestamp 0 and at every new broadcast, i increments the timestamp by 1. The algorithm can extract these two parameters from m by respectively calling the functions $source(m)$ and $ts(m)$.

Process i keeps the following local variables:

- $correct_i$: the set of processes considered correct by process i;
- $last_i[n]$: an array of n elements to keep the last messages delivered by i ($last_i[j]$ is the last message broadcast by j that was delivered by i);
- ack_set_i: a set with all pending acknowledgement messages of process i. For each message $\langle TREE, m\rangle$ received by i from process j and retransmitted to process k, an element $\langle j, k, m\rangle$ is added to this set; The symbol $\perp$ represents a null element. The asterisk is used as a wildcard. For instance, $\langle j, *, m\rangle$ means all pending $acks$ for a message m received from process j and re-sent to any other process;
- $pending_i$: list of the messages received by i that where not delivered yet because they are "out of order" with regard to their timestamp, i.e., $ts(m) > ts(last_i(source(m))+1$;
- $history_i$: the history of messages that were already broadcast by i. This set is used to prevent sending the same message to the same cluster more than once. $\langle j, m, h\rangle \in history_i$ indicates that the message m received from process j was already sent by i to the clusters $c_{i,s}$ for all $s \in [1, h]$.

5.2. Algorithm description

Process i broadcasts a message by calling the BROADCAST(m) function. Line 7 ensures that a new *broad-*

Algorithm 1 Reliable broadcast - process i

1: $last_i[n] \leftarrow \{\perp, ..., \perp\}$
2: $ack\_set_i \leftarrow \emptyset$
3: $correct_i \leftarrow \{0, ..., n-1\}$
4: $pending_i \leftarrow \emptyset$
5: $history_i \leftarrow \emptyset$

6: **procedure** BROADCAST(message m)
7: **wait until** $ack\_set_i \cap \{\langle \perp, *, last_i[i] \rangle\} = \emptyset$
8: $last_i[i] \leftarrow m$
9: DELIVER(m)
10: BROADCAST_TREE($\perp, m, log_2 n$)

11: **procedure** BROADCAST_TREE(process j, message m, integer h)
12: $start \leftarrow 0$
13: **if** $\exists x : \langle j, m, x \rangle \in history_i$ **then**
14: $start \leftarrow x$
15: $history_i \leftarrow history_i \backslash \{\langle j, m, x \rangle\}$
16: $history_i \leftarrow history_i \cup \{\langle j, m, max(start, h) \rangle\}$
17: **if** $start < h$ **then**
18: **for all** $s \in [start + 1, h]$ **do**
19: BROADCAST_CLUSTER(j, m, s)

20: **procedure** BROADCAST_CLUSTER(process j, message m, integer s)
21: $sent \leftarrow false$
22: **for all** $k \in c_{i,s}$ **do**
23: **if** $sent = false$ **then**
24: **if** $\langle j, k, m \rangle \in ack\_set_i$ **and** $k \in correct_i$ **then**
25: $sent \leftarrow true$
26: **else if** $k \in correct_i$ **then**
27: SEND($\langle TREE, m \rangle$) to p_k
28: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle j, k, m \rangle\}$
29: $sent \leftarrow true$
30: **else if** $\langle j, k, m \rangle \notin ack\_set_i$ **then**
31: SEND($\langle DELV, m \rangle$) to p_k

32: **procedure** CHECK_ACKS(process j, message m)
33: **if** $j \neq \perp$ **and** $ack\_set_i \cap \{\langle j, *, m \rangle\} = \emptyset$ **then**
34: SEND($\langle ACK, m \rangle$) to p_j

35: **procedure** HANDLE_MESSAGE(process j, message m)
36: $pending_i \leftarrow pending_i \cup \{m\}$
37: **while** $\exists l \in pending_i : source(l) = source(m)$
38: $\wedge (ts(l) = ts(last_i[source(l)]) + 1$
39: **or** $last_i[source(l)] = \perp \wedge ts(l) = 0)$ **do**
40: $last_i[source(l)] \leftarrow l$
41: $pending_i \leftarrow pending_i \backslash \{l\}$
42: DELIVER(l)
43: **if** $source(m) \notin correct_i$ **then**
44: BROADCAST_TREE($j, last_i[source(m)], log_2 n$)

45: **upon** `receive` $\langle TREE, m \rangle$ **from** p_j
46: HANDLE_MESSAGE(m)
47: BROADCAST_TREE($j, m, cluster_i(j) - 1$)
48: CHECK_ACKS(j, m)

49: **upon** `receive` $\langle DELV, m \rangle$ **from** p_j
50: HANDLE_MESSAGE(m)

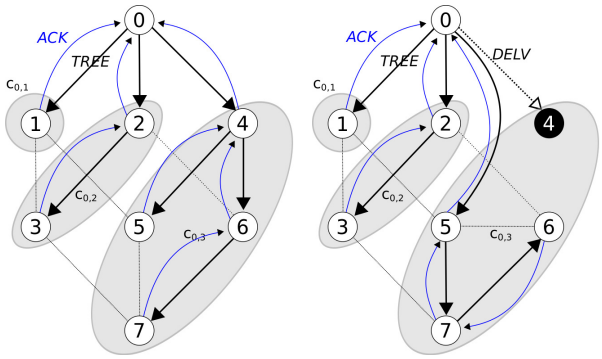
51: **upon** `receive` $\langle ACK, m \rangle$ **from** p_j
52: **for all** $k = x : \langle x, j, m \rangle \in ack\_set_i$ **do**
53: $ack\_set_i \leftarrow ack\_set_i \backslash \{\langle k, j, m \rangle\}$
54: CHECK_ACKS(k, m)

55: **upon notifying** `crash`(j)
56: $correct_i \leftarrow correct_i \backslash \{j\}$
57: **for all** $p = x, m = y : \langle x, j, y \rangle \in ack\_set_i \cap \{\langle *, j, * \rangle\}$ **do**
58: BROADCAST_CLUSTER($p, m, cluster_i(j)$)
59: $ack\_set_i \leftarrow ack\_set_i \backslash \{\langle p, j, m \rangle\}$
60: CHECK_ACKS(p, m)
61: **if** $last_i[j] \neq \perp$ **then**
62: BROADCAST_TREE($j, last_i[j], log_2 n$)

63: **upon notifying** `up`(j)
64: $correct_i \leftarrow correct_i \cup \{j\}$

cast starts only after the previous one has been completed, i.e., there is no pending *acks* for the $last_i[i]$ message. Note that some processes might not have received the previous message yet because of false suspicions. Then, the received message m is locally delivered to i (line 9) and, by calling the function BROADCAST_TREE (line 10), i forwards m to its neighbors in the VCube. To this end, it calls, for each cluster $s \in [1, log_2 n]$, the function BROADCAST_CLUSTER that sends a `TREE` message to the first process k which is correct in the cluster (line 27). To those processes that are not correct, i.e, suspected of being crashed, and placed before k in the cluster, a `DELV` message (line 31) is sent to them. Notice that in both cases, the messages are sent provided i has not already forwarded m, received from j, to k. For every sent `TREE` message the corresponding *ack* is included in the list of pending *acks* (line 28).

Let's consider the 3-VCube topology of Figure 2. Figure 2(a) shows a fault-free scenario where process 0 (p_0) broadcasts a message. After delivering the message to itself,



(a) No failures (b) p_4 suspected

Figure 2. Reliable broadcast - process 0 (p_0)

p_0 sends a copy of the message to p_1, p_2, and p_4, which are neighbors of p_0 and the first correct process on each of i's clusters.

Upon reception of a message $\langle TREE, m\rangle$ from process j (line 45), process i calls the function HANDLE_MESSAGE. In this function, m is added to the set of pending messages and then all pending messages which were broadcast by the same process that broadcast m ($source(m)$) are delivered in increasing order of timestamps, provided no message is missing in the sequence of timestamps (lines 36 - 42). In the same HANDLE_MESSAGE function, if i suspects that $source(m)$ failed, it restarts the broadcast of $last_i[source(m)]$ (line 44) to ensure that every correct process receives the message even if $source(m)$ crashed in the middle of the broadcast. Otherwise, by calling the function BROADCAST_TREE with parameter $h = cluster_i(j) - 1$ (line 47), m is forwarded to all neighbors of i in each subcluster of i that should receive m. Figure 2(a) shows the forwarding of m from p_4 to p_5.

If process i is a leaf in the spanning tree of the broadcast ($cluster_i(j)-1 = 0$) or if all neighbors of i (i.e., children of i in the tree) that should receive the message are suspected of being crashed, i sends an ACK message to the process which sent m to it, by calling function CHECK_ACKS (line 34).

If process i receives a $\langle DELV, m\rangle$ message from j (line 49), it means that j falsely suspects i of being crashed and has decided to trust another process with the forwarding of the message to the rest of the tree. Therefore i can simply call the HANDLE_MESSAGE function to deliver the message and does not need to call BROADCAST_TREE.

Whenever i receives a message $\langle ACK, m\rangle$, it removes the corresponding ack from set of pending $acks$ (line 53) and, by calling the function CHECK_ACKS, if there are no more pending *acks* for message m, i sends an ACK message to the process j which sent m to it (line 34). If $j = \perp$, the ACK message has reached the process that has broadcast m ($source(m)$) and the ACK message does not need to be forwarded.

The detection of the failure of process j is notified to i (crash(j)). It is worth pointing out that this detection might be a false suspicion. Three actions are taken by i upon receiving such a notification: (1) update of the set of processes that it considers correct (line 56); (2) removal from the set of pending *acks* of those *acks* whose related message m has been retransmitted to j (line 59); (3) re-sending to k, the next neighbor of j in the cluster of j (if k exists), of those messages previously sent to j. The re-sending of these messages triggers the propagation of messages over a new spanning tree (line 58). For instance, in Figure 2(b), after the notification of the failure of p_4, p_0 sends message m to p_5 since the latter is the next fault-free neighbor of p_4 in $c_{0,3} = \{4, 5, 6, 7\}$ (*cluster* $s = 3$) The message is then propagated to the other correct processes of the cluster, i.e., processes p_6 and p_7. Notice that if p_4 is considered faulty by p_0 before the start of the broadcast, p_0 sends a DELV message to p_4 in order to ensure the reception and handle of m by p_4. Finally, in case of crash of j, i has to re-broadcast the last message broadcast by j (line 62). Notice that, in this case, the $history$ variable is used in order to prevent i from re-rebroadcasting the message to those clusters that i has already sent the same message.

If VCube detects that it had falsely suspected process j, it corrects its mistake and notifies i which then includes j in its set of correct processes (line 64).

5.3. Proof of correctness

In this section we will prove that Algorithm 1 implements a reliable broadcast.

Lemma 1. *Algorithm 1 ensures the validity property of reliable broadcast.*

Proof. If a process i broadcasts a message m, the only way that i would not deliver m is if i waits forever on line 7. This wait is interrupted when the set ack_set_i contains no more pending acknowledgements related to the message $last_i[i]$ previously broadcast by i.

For any process j that i sent $last_i[i]$ to, i added a pending ack in ack_set_i (line 28). If j is correct, then it will eventually answer with an ACK message (line 34) and i will remove $\langle \perp, j, last_i[j]\rangle$ from ack_set_i on line 53. If j is faulty, then i will eventually detect the crash and remove the pending ack on line 59.

As a result, all of the pending acks for $last_i[i]$ will eventually be removed from ack_set_i and i will deliver m on line 9.

Line 9 then ensures that i will deliver the message before broadcasting it. □

Lemma 2. *For any processes i and j, the value of $ts(last_i[j])$ only increases over time.*

Proof. For the sake of simplicity, we take the convention that $ts(\perp) = -1$. The $last_i$ array is only modified on lines 8 and 40.

The first case can only happen when i broadcasts a new message m, and since timestamps of new messages sent by a same processes have to be increasing, $ts(m) > ts(last_i[i])$. When i calls the $broadcast$ procedure with m, $ts(last_i[i])$ will therefore increase on line 8.

The other way for $last_i$ to be modified is on line 40. $last_i[source(l)]$ will then be updated with message l if $last_i[source(l)] = \perp$ and $ts(l) = 0$ (and therefore $ts(last_i[source(l)]) = -1 < ts(l)$), or if $ts(l) = ts(last_i[source(l)]) + 1$. It follows that $last_i[source(l)$ is only updated if the new value of $ts(last_i[source(l)])$ would be superior to the old one. □

Lemma 3. *Algorithm 1 ensures the integrity property of reliable broadcast.*

Proof. Processes only deliver a message if they are broadcasting it themselves (line 9) or if the message is in their $pending_i$ set (line 42). Messages are only added to the $pending_i$ set on line 36, after they have been received from another process. Since the links are reliable and do not create

messages, it follows that a message is delivered only if it was previously broadcast (there is no creation of messages).

To show that there is no duplication of messages, let us consider two cases:

- $\mathbf{source(m) = i}$. Process i called the $broadcast$ procedure with parameter m. As proved in Lemma 1, i will deliver m on line 9. Since the $broadcast$ procedure is only called once with a given message, the only way that i would deliver m a second time is on line 42. Since $last_i[i]$ was set to m on line 8, it follows from Lemma 2 that m will never qualify to pass the test on lines 37 – 39.
- $\mathbf{source(m) \neq i}$. Process i is not the emitter of message m, and did not call the $broadcast$ procedure with m. Therefore the only way for i to deliver m is on line 42. Before i delivers m for the first time, it sets $last_i[source(m)]$ to m on line 40. It then follows from Lemma 2 that m will never again qualify to pass the test on lines 37 – 39, and therefore i can deliver m at most once.

□

Lemma 4. *Algorithm 1 ensures the agreement property of reliable broadcast.*

Proof. Let m be a message broadcast by a process i. We consider two cases:

- **i is correct**. It can be shown by induction that every correct process receives m.
  As a basis of the induction, let us consider the case where $n = 2$ and $P = \{i, j\}$. It follows that $c_{i,1} = \{j\}$. Therefore i will send m to j on line 31 if i suspects j or on line 27 otherwise. If j is correct, it will eventually receive m since the links are reliable, and will deliver m on line 42. i will also deliver m, by virtue of the validity property.
  We now have to prove that if every correct process receives m for $n = 2^k$, it is also the case for $n = 2^{k+1}$. The system of size 2^{k+1} can be seen as two subsystems $P_1 = \{i\} \cup \bigcup_{x=1}^{k} c_{i,x}$ and $P_2 = c_{i,k+1}$ such that $|P_1| = |P_2| = 2^k$.
  The $broadcast\_tree$ and $broadcast\_cluster$ procedures ensure that for every $s \in [1, k+1]$, i will send m to at least one process in $c_{i,s}$. Let j be the first process in $c_{i,k+1}$. If j is correct, it will eventually receive m. If j is faulty and i detected the crash prior to the broadcast, i will send the message to j anyway in case it is a false suspicion (line 31) but it will also send it to another process in $c_{i,k+1}$ as a precaution (line 27). i will keep doing so until it has sent the $TREE$ message to a non-suspected process in $c_{i,k+1}$, or until it has sent the message to all the processes in $c_{i,k+1}$.
  If j is faulty and i only detects the crash after the broadcast, the $broadcast\_cluster$ procedure will be called again on line 58, which ensures once again that i will send the message to a non-suspected process in $c_{i,k+1}$. As a result, unless all the processes in $c_{i,k+1}$ are faulty, at least one correct process in $c_{i,k+1}$ will eventually receive m. This correct process will then broadcast m to the rest of the P_2 subsystem on line 47. Since a correct process broadcasts m in both subsystems P_1 and P_2, and since both subsystems are of size 2^k, it follows that every correct process in P will eventually receive m.

- **i is faulty**. If i crashes before sending m to any process, then no correct process delivers m and the agreement property is verified. If i crashes after the broadcast is done, then everything happens as if i was correct. If i crashes after sending m to some processes and a correct process j receives m, then j will eventually detect the failure of i. If j detects the crash before receiving m, when it receives m it will restart a full broadcast of m on line 44. If j only detects the crash of i after receiving m, it will also restart a full broadcast of m on line 62. Since j is correct, every correct process will eventually receive m.

□

Theorem 1. *Algorithm 1 implements a reliable broadcast.*

Proof. The proof follows directly from Lemmas 1, 3, and 4. □

6. Performance Discussion

The goal of exploiting the VCube overlay in our solution is to provide an efficient broadcast where each process sends at most $log_2 n$ messages. However, this complexity cannot be ensured at all times in an asynchronous system where false suspicions can arise. Algorithm 1 aims to take advantage of the VCube whenever possible while still ensuring the properties of a reliable broadcast despite false suspicions.

In the best case scenario where no process is ever suspected of failure, each process will send at most one message per cluster (line 27). Therefore $n - 1$ $TREE$ messages will be sent in total (since no process will be sent the same message twice) with no single process sending more than $log_2 n$ messages. This is the example presented in Figure 2(a).

If a process other than the source of the broadcast is suspected before the broadcast, there will be $n - 2$ $TREE$ messages and one $DELV$ message sent. A single process might send up to $log_2 n$ $TREE$ messages plus one $DELV$ message per suspected process. This is the example of Figure 2(b).

If the source of the broadcast suspects everyone else, then it will send $n - 1$ $DELV$ messages. In this case, Algorithm 1 is equivalent to a *one-to-all* algorithm where one process sends the message directly to all others, losing, thus, the advantages of tree topology properties, such as scalability.

The main cost of suspicions lies in the fact that when a process is suspected, its last broadcast must be resent. This is the purpose of lines 44 and 62. Such a re-broadcast is

an unavoidable consequence of the existence of false suspicions, necessary in order to ensure the agreement property of reliable broadcast.

Note that the fact that the information about a node failure is false or true has no difference in the impact on the performance of the broadcast algorithm in terms of message complexity.

7. Conclusion and Future Work

This article presented a reliable broadcast algorithm for message-passing distributed systems prone to crash failures on asynchronous environments. It tolerates up to n-1 failures. For broadcasting a message, the algorithm dynamically builds a spanning tree over a virtual hypercube topology provided by the underlying monitor system VCube. In case of failure, the tree is dynamically reconstructed. To this end, the VCube provides information about node failures. However, as the system is asynchronous, it can make mistake falsely suspecting no faulty nodes. Such false suspicions are tolerated by the algorithm by sending special messages to those processes suspected of having failed. In summary, whenever possible, the algorithm exploits the hypercube properties offered by the VCube while ensuring the properties of the reliable broadcast, even in case of false suspicions.

As future work, we intend to implement our algorithm and conduct extensive simulation experiments in order to compare its performance in terms of latency and number of messages in different scenarios with and without failure of nodes as well as false suspicions.

References

[1] S. Bonomi, A. Del Pozzo, and R. Baldoni, "Intrusion-tolerant reliable broadcast," Sapienza Università di Roma,, Technical Report, 2013.

[2] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," S. Mullender, Ed. New York, NY, USA: ACM Press, 1993, ch. Distributed systems, pp. 97–145.

[3] R. Guerraoui and L. Rodrigues, Eds., *Introduction to Reliable Distributed Programming*. Berlin, Germany: Springer-Verlag, 2006.

[4] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[5] F. B. Schneider, D. Gries, and R. D. Schlichting, "Fault-tolerant broadcasts," *Sci. Comput. Program.*, vol. 4, no. 1, pp. 1–15, May 1984.

[6] P. Fragopoulou and S. Akl, "Edge-disjoint spanning trees on the star network with applications to fault tolerance," *IEEE Trans. Comput.*, vol. 45, no. 2, pp. 174 –185, Feb. 1996.

[7] K. Kim, S. Mehrotra, and N. Venkatasubramanian, "FaReCast: Fast, reliable application layer multicast for flash dissemination," in *ACM/IFIP/USENIX 11th International Conference on Middleware*, ser. Middleware'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 169–190.

[8] P. Ramanathan and K. Shin, "Reliable broadcast in hypercube multicomputers," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1654–1657, 1988.

[9] M. Raynal, J. Stainer, J. Cao, and W. Wu, "A simple broadcast algorithm for recurrent dynamic systems," in *Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, ser. AINA '14, 2014, pp. 933–939.

[10] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[11] E. P. Duarte, Jr., L. C. E. Bona, and V. K. Ruoso, "VCube: A provably scalable distributed diagnosis algorithm," in *5th Work. on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA'14. Piscataway, USA: IEEE Press, 2014, pp. 17–22. [Online]. Available: http://dx.doi.org/10.1109/ScalA.2014.14

[12] E. P. Duarte Jr. and T. Nanya, "A hierarchical adaptive distributed system-level diagnosis algorithm," *IEEE Trans. Comput.*, vol. 47, no. 1, pp. 34–45, Jan. 1998.

[13] L. A. Rodrigues, L. Arantes, and E. P. Duarte Jr., "An autonomic implementation of reliable broadcast based on dynamic spanning trees," in *10th European Dependable Computing Conference*, ser. EDCC'14, 2014, pp. 1–12.

[14] J. Wu, "Optimal broadcasting in hypercubes with link faults using limited global information," *J. Syst. Archit.*, vol. 42, no. 5, pp. 367–380, 1996.

[15] J. Liebeherr and T. Beam, "HyperCast: A protocol for maintaining multicast group members in a logical hypercube topology," in *Networked Group Communication*, ser. LNCS, L. Rizzo and S. Fdida, Eds. Springer Berlin Heidelberg, 1999, vol. 1736, pp. 72–89.

[16] J. Leitão, J. Pereira, and L. Rodrigues, "HyParView: A membership protocol for reliable gossip-based broadcast," in *DSN*, 2007, pp. 419–429.

[17] M. Ferreira, J. Leitão, and L. Rodrigues, "Thicket: A protocol for building and maintaining multiple trees in a p2p overlay," in *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems*, Oct. 2010, pp. 293–302.

[18] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.